

CMSC 430 Practice Midterm

In grammars, capital letters represent nonterminals,
lower case letters represent terminals.
One sentence answers are sufficient for the “essay” questions.

1. (6 points) Compiler front end.
 - (a) What is the primary function of the scanner and what computational mechanism is used to accomplish it?
 - (b) What is the primary function of the parser and what computational mechanism is used to accomplish it?
 - (c) How do you decide what should be handled by the scanner versus the parser? (Hint: think of the complexity of languages)

2. (20 points) Scanner construction.
 - (a) Construct a regular expression for recognizing all non-empty strings composed of the letters **a** and **b** that do not end in **b**.
 - (b) Convert the regular expression to an NFA using the construction algorithm given in class.
 - (c) Convert the NFA to a DFA (show the sets of NFA states for each DFA state).
 - (d) Minimize the DFA (show the sets of DFA states for each minimized DFA state).

3. (16 points) Consider the following grammar:

$$E \rightarrow E + E \mid a$$

- (a) When is a grammar ambiguous?
- (b) Show that the grammar is ambiguous for the string $a + a + a$
- (c) What happens if you write a recursive-descent parser for this grammar?
- (d) Fix the grammar to avoid this problem.

4. (16 points) Consider the following grammar:

$$\begin{aligned} S &\rightarrow ABd \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow b \mid cA \end{aligned}$$

- (a) Calculate FIRST, FOLLOW for S, A, B:
- (b) Construct its recursive-descent parser (with lookahead), given the following functions:

```
tok; // current token

match(x) { // matches token
  if (tok != x) // if wrong token
    error(); // exit with error
  tok = getToken(); // get new token
}

parser() {
  tok = getToken(); // initialize
  S(); // start symbol
  match("$"); // match EOF
}
```

5. (6 points) Consider the following ACTION/GOTO tables:

State	Action		Goto	
	a	\$	A	B
0	shift 1	reduce $B \rightarrow \epsilon$	2	3
1	shift 3	reduce $A \rightarrow a$	0	
2		accept	3	0
3	shift 1	accept		1

Show the contents of the stack and input buffer for the shift-reduce parse of "a", assuming State 0 is the start state:

6. (20 points) Consider the following augmented grammar:

$$\begin{array}{l|l} P1 & S \rightarrow E \\ P2 & E \rightarrow E + E \\ P3 & \quad \quad \quad | a \end{array}$$

- (a) Derive the canonical sets of LR(1) items
- (b) Build the LR(1) parse table
- (c) For each shift/reduce or reduce/reduce conflict you find (if any), describe what would happen for the different ways to resolve the conflict(s).

7. (16 points) Consider the following sets of LR(1) items in the states of a LR(1) parser:

State 0: $[A \rightarrow \bullet a, b]$ $[A \rightarrow a \bullet, c]$ $[B \rightarrow a \bullet, b]$	State 2: $[A \rightarrow \bullet a, c]$ $[A \rightarrow a \bullet, b]$ $[B \rightarrow a \bullet, a]$
---	---

State 1: $[A \rightarrow \bullet a, a]$ $[A \rightarrow \bullet a, b]$ $[B \rightarrow a \bullet, b]$	State 3: $[A \rightarrow \bullet a, b]$ $[B \rightarrow \bullet a, b]$
---	---

- (a) Find all conflicts, listing state, pair of LR(1) items, and lookahead(s) causing conflict.
- (b) List states that would be merged in a LALR(1) parser.
- (c) List additional conflicts in the LALR(1) parser, if any.
- (d) What are the advantages of LALR over LR parsers?

8. Syntax-directed translation.

- (a) What are the main differences between attribute grammars and ad-hoc, syntax-directed translation?
- (b) Name one advantage and disadvantage of each.

9. Type checking.

- (a) Consider the following C code.

```
int foo(char *x, int bar[4]);
```

What is the type expression for `foo`?

- (b) Consider the following grammar productions. Assume you have an attribute `const.odd` which is set to either true or false. How can we then extend the type checker to determine whether an expression is odd (or even)?

$$E \rightarrow \text{const} \quad \{ E.\text{odd} = ?? \}$$

$$| E_1 + E_2 \quad \{ E.\text{odd} = ?? \}$$

Give the actions for both productions.

10. Symbol table, run-time environments.

Consider the following program in a lexically-scoped language such as C.

```
int x,y;
int f( int p) {
  int y,z;
  {
    int i,j;
    /* Point A */
  }
  /* Point B */
}
/* Point C */
```

- (a) How can the compiler organize symbol table information at compile time to handle nested scoping? Draw the logical state of the symbol table(s).
- (b) What symbols are visible at points A and B?
- (c) Name some information stored for each procedure at run time.
- (d) How does the compiler allocate storage for `x`, `y`, and `z` at run time?

11. Internal representations.

Use the following 3-address and Java stack code instructions for answering questions on the midterm.

3-addr Instruction	Effect
load R1 x	$R1 \leftarrow x$
store x R1	$x \leftarrow R1$
add R1 R2 R3	$R1 \leftarrow R2 + R3$
sub R1 R2 R3	$R1 \leftarrow R2 - R3$
mult R1 R2 R3	$R1 \leftarrow R2 * R3$
neg R1 R2	$R1 \leftarrow -(R2)$

Java Stack Code	Effect
nop	none
ldc_int c	push constant <i>c</i> onto stack
iload index(x)	push local variable <i>X</i> onto stack
istore index(x)	pop stack, store in local variable <i>X</i>
iadd	pop 2 elems off stack, add, push
isub	pop 2 elems off stack, subtract, push
imult	pop 2 elems off stack, multiply, push
ineg	pop stack, negate, push
goto L	jump to handle <i>L</i>
ifeq L	pop stack, jump to handle <i>L</i> if zero
if_icmpeq L	pop 2 elems, jump to <i>L</i> if equal
if_icmpgt L	pop 2 elems, jump to <i>L</i> if 1st greater
dup	duplicate top of stack
pop	pop top of stack
swap	swap top two positions of stack

Consider the code $y = (x-y)+(x*z)$

- (a) Translate it into an AST.
- (b) Translate it into 3-address code.
- (c) Translate it into Java stack code.
- (d) What representation(s) are more compact? Why?
- (e) What representation(s) are easier for program transformations? Why?